IN THE SPECIFICATION:

Kindly amend the specification as indicated below. In addition to the required underlining and bracketing or strike through text, where Applicants believed it helpful to indicate the exact location within a paragraph to amend, text has been marked with bold text characters. Please disregard this bold text formatting when making the following changes.

Please, amend the title of the application as follows:

B1

~~SYSTEM AND METHOD FOR~~ INSTRUCTION LEVEL MULTITHREADING IN AN EMBEDED PROCESSOR USING ZERO-TIME CONTEXT SWITCHING

On page 7, in the paragraph beginning at line 7, change "be" to "been" as follows:

B2

An embedded processor is a processor that is used for specific functions. Embedded processors generally have some memory and peripheral functions integrated on-chip. Conventional embedded processors have not [[be]] been capable of operating using multiple hardware threads.

On page 8, in the paragraph beginning at line 1, change "are" to "is" and add a comma after (MIPS) as follows:

B3

A problem with conventional pipelined processors is that because the speed of CPUs [[are]] is increasing, it is increasingly difficult to fetch instruction opcodes from flash memory without having wait-states or without stalling the instruction pipeline. A faster memory, e.g., static RAM (SRAM) could be used to increase instruction fetch times but requires significantly more space and power on the embedded processor. Some conventional systems have attempted to overcome this problem using a variety of techniques. One such technique is to fetch and execute from flash memory. This technique would limit the execution speed of conventional processors, e.g., to 40 million instructions per second (MIPS), which is unacceptable in many applications.

On page 9, in the paragraph beginning at line 1, change "is" to "be" as follows:

B4

A third technique is to use flash memory and SRAM cache. When the program reference is within the SRAM, then full speed execution is possible, but otherwise a cache miss occurs that leads to a long wait during the next cache load. Such a system results in unpredictable and undeterministic execution time that is generally unacceptable for processors that are real-time constrained. The real-time constraints are imposed by the requirement to meet the timing required by standards such as IEEE 802.3 (Ethernet), USB, HomePNA 1.1 or SPI (Serial Peripheral Interface). These standards require that a response [[is]] be generated within a fixed amount of time from an event occurring.

On page 11, in the paragraph beginning at line 1, add a dash between "fetch" and "switching" as follows:

B5

Figure 4 is an illustration of a multithreaded **fetch-switching** pipeline according to one embodiment of the present invention.

On page 13, in the paragraph beginning at line 1, add a dash between "issue" and "switching" as follows:

B6

Figure 14 is an illustration of a multithreaded **issue-switching** pipeline according to one embodiment of the present invention.

On page 15, in the paragraph beginning at line 6, add a period after "... interleave with thread C" as follows:

B7

The multithreading capability of the present invention permits multiple threads to exist in the pipeline concurrently. Figure 2 is an illustration of an interrupt response in a multithreaded environment. Threads A and B are both hard-real-time (HRT) threads which have stalled pending interrupts A and B respectively. Thread C is the main code thread and is non-real-time (NRT). When interrupt A occurs, thread A is resumed and will **interleave with thread C.** Thread C no longer has the full pipeline throughput since it is NRT. When interrupt B occurs thread B is resumed, and, being of the same priority as thread A, will interleave down the pipeline, thread C is now completely stalled. The main code – thread C will continue executing only when the HRT threads are no longer using all of the pipeline throughput.

On page 16, in the paragraph beginning at line 15, add a dash between "fetch" and "switching" as follows:

B8

As described above, a feature of the present invention is the ability to switch between thread contexts with no overhead. A zero overhead context switch can be achieved by controlling which program-counter the fetch unit uses for instruction fetching. Figure 4 is an illustration of a multithreaded **fetch-switching** pipeline according to one embodiment of the present invention. The processor will function with information from different threads at different stages within the pipeline as long as all register accesses relate to the correct registers within the context of the correct thread.

On page 18, in the paragraph beginning at line 9, add a dash between "general" and "purpose" as follows:

B9

Figure 6 is an illustration of an example of a per-thread context according to one embodiment of the present invention. The context in this example includes 32 **general-purpose** registers, 8 address registers and a variety of other information as illustrated. The type of data that is stored as part of a thread's context may differ from that illustrated in Figure 6.

On page 18, in the paragraph beginning at line 17, change "quanta" for "quantum" as follows:

B10

The present invention is an instruction level multithreading system and method that takes advantage of the zero-time context switch to rapidly (as frequently as every instruction) switch between two or more contexts. The amount of time that each context executes for is called a *quantum*. The smallest possible [[quanta]] quantum is one clock cycle, which may correspond to one instruction. A [[quanta]] quantum may also be less than one instruction for multi-cycle instructions (i.e., the time-slice resolution is determined solely by the quantum and not the instruction that a thread is executing). A detailed description of the allocation and scheduling is described below and is described in U.S. Patent Application No. 09/748,098 that is incorporated by reference herein in its entirety.

On page 21, in the paragraph beginning at line 1, replace "is" with "are" as follows:

B11

Some of the benefits of using either strict scheduling or semi-flexible scheduling [[is]] are that the allocation of execution time for each HRT thread is set and therefore the time required to execute each thread is predictable. Such predictability is important for many threads since the thread may be required to complete execution within a specific time period. In contrast, the interrupt service routine described above with reference to conventional systems does not ensure that the hard real time threads will be completed in a predictable time period.

On page 23, in the paragraph beginning at line 1, add a dash between "hard" and "real" and replace "which" with "that" as follows:

B12

Although the table reserves the instruction slots for the **hard-real**-time tasks this does not mean that other non-real time tasks cannot also execute in that instruction slot. For example, thread C may be idle most of the time. For example, if thread C represents a 115.2 kbps UART, then it only needs deterministic performance when it is sending or receiving data. There is no need for it to be scheduled when it is not active. All empty instruction slots, and those instruction slots that [[which]] are allocated to a thread that is not active can be used by the scheduler for non-real-time threads.

On page 24, in the paragraph beginning at line 8, replace "an" with "a" and add a dash between "hard" and "real" as follows:

B13

Multiple levels of priority are supported for non-real-time threads. A low priority thread will always give way to higher priority threads. The high level priority allows the implementation of [[an]] a real-time operating system (RTOS) in software by allowing multi-instruction atomic operations on low-priority threads. If the RTOS kernel NRT thread has a higher priority than the other NRT threads under its control then there is a guarantee that no low priority NRT threads will be scheduled while the high priority thread is active. Therefore the RTOS kernel can perform operations without concern that it might be interrupted by another NRT thread.

On page 25, in the paragraph beginning at line 7, replace "and is" with "as" and add dashes between "hard" and "real" as follows:

B14

The present invention includes hardware support for running multiple software threads and automatically switching between threads [[and is]] as described below. This multi-threading support includes a variety of features including **real-time** and non-**real-time** task scheduling, inter-task communication with binary and counting semaphores (interrupts), fast interrupt response and context switching, and incremental linking.

On page 25, in the paragraph beginning at line 16, remove a dash between "context" and "switching" as follows:

B15

By including the multi-threading support in the embedded processor core the overhead for a context switch can be reduced to zero. A zero-time context-switch allows context switching between individual instructions. Zero-time context[[-]]switching can be thought of as time-division multiplexing of the core.

On page 25, in the paragraph beginning at line 23, add "fetch logic" as follows:

B16

In one embodiment of the present invention fetch logic can fetch code from both SRAM and flash memory, even when the flash memory is divided into multiple independent blocks. This complicates the thread scheduling of the present invention. In the present invention, each memory block is scheduled independently of the overall scheduling of threads. Figure 8 is an illustration of thread fetching logic with two levels of scheduling according to one embodiment of the present invention.

On page 26, in the paragraph beginning at line 8, add dashes between "hard" and "real" and "real" and "time," delete "fetching" and add "fetched" as follows:

B17

In one embodiment of the present invention the instructions that are fetched can be stored in multiple types of memory. For example, the instructions can be stored in SRAM and flash memory. As described above, accessing data, e.g., instructions, from SRAM is significantly faster than accessing the same data or instructions from flash memory. In this embodiment, it is preferable to have **hard-real-time** threads be fetched in a single cycle so all instructions for hard-real-time threads are stored in the

B17

SRAM. In contrast, [[fetching]] instructions fetched from non-real-time threads can be stored in either SRAM or flash memory.

On page 27, in the paragraph beginning at line 10, add dashes between "hard" and "real" and "real" and "time," replace "back" with "bank," remove the extra period after "executing," and insert two paragraph breaks as follows:

B18

Figure 9 is an illustration of the HRT thread selector 802 according to one embodiment of the present invention. As indicated above, the shadow SRAM 820 provides a single cycle random access instruction fetch. Since **hard-real-time**(HRT) threads require single cycle determinism in this embodiment such HRT threads may execute only from SRAM. The HRT thread controller 802 includes a bank selector 902 that allows the choice of multiple HRT schedule tables. The [[back]] bank selector determines which table is in use at any particular time. **(Paragraph break)**

The use of multiple tables permits the construction of a new schedule without affecting HRT threads that are already **executing**.[[.]] A counter 904 is used to point to the time slices in the registers in the HRT selector 802. The counter will be reset to zero when either the last entry is reached or after time slice 63 is read. The counter 904 is used in conjunction with the bank selector 902 to identify the thread that will be fetched by the shadow SRAM in the following cycle. If the identified thread is active, e.g., not suspended, then the program counter (PC) of the identified thread is obtained and is used as the address for the shadow SRAM in the following cycle. For example, with respect to Figure 9, if based upon the bank selector 902 and the counter 904 time slice number 1 is identified, the thread identified by this time slice represents the thread that will be fetched by the SRAM in the following cycle. **(Paragraph break)**

The output of the block described in Figure 9 is a set of signals. Eight signals are used to determine which of the eight threads is to be fetched. Of course this invention is not limited to controlling only eight threads. More signals could be used to control more threads. One signal is used to indicate that no HRT thread is to be fetched.

On page 33, in the paragraph beginning at line 1, remove the comma after "thread" as follows:

B19

Other processors have a separate operating system (OS) kernel running on each hardware thread[[,]] that is responsible for saving and restoring the state of that thread. This may not be adequate if code from an existing RTOS is to be used to control hardware threads.

On page 33, in the paragraph beginning at line 8, add dashes between "memory" and "to" and "to" and "memory" as follows:

B20

One type of instruction set that can be used with the present invention is a **memory-to-memory** instruction set, with one general source (memory or register), one register source, and one general destination. The invention allows one thread to set its general source, general destination, or both, to use the thread state of another thread. Two fields in the *processor status word*, *source thread*, and *destination thread*, are used, and override the normal thread ID for the source and/or destination accesses to registers or memory.

On page 33, in the paragraph beginning at line 24, add a dash between "above" and "described" as follows:

B21

In another embodiment of the present invention, the **above-described** multithreading system can be used with more powerful pipeline structures. Figure 14 is an illustration of a multithreaded issue-switching pipeline according to one embodiment of the present invention. In conventional pipeline processing environments the fetch and decode stages result in the same output regardless of when the fetch and decode operations occur. In contrast the issue stage is data dependant, it obtains the data from the source registers, and therefore the result of this operation depends upon the data in the source registers at the time of the issue operation. In this embodiment of the present invention the thread-select decision is delayed to the input of the issue stage.